Copyright (C)2015 ARRL, All Rights Reserved

Scotty Cowling, WA2DFI

PO Box 26843, Tempe, AZ 85285: **scotty@tonks.com**

Hands On SDR

In this installment we take a look under the hood of a stand-alone software defined radio field programmable gate array (FPGA), to see just how logic gates and registers build a useful radio.

In this second column, I will show you how to set up an FPGA coding environment with free development tools, walk you through the code of an SDR design example and show you how to compile the example code and run it on real hardware. All of this will be very hands-on. We will, of necessity, cover some SDR theory, but we will be taking the background mathematics as a given and instead focus on how we implement the math functions inside the FPGA. We will be reviewing the design at what is called the RTL (register-transfer level) in an HDL (hardware description language) known as *Verilog*. The other main language used by FPGA designers is called *VHDL*, but we will not cover that here. Don't be intimidated by all of these TLAs (three letter acronyms); professionals use these by the boatload to make themselves SMK (sound more knowledgeable). Okay, Okay, so I made up that last one, but you get the idea. Don't I SMK already?

Is this for me?

As with each of these columns, limited space begs the questions: "What do I need to know?" and "What equipment do I need?"

You will need a basic working knowledge of the *Verilog* hardware description language. You should be able to pick this up easily by following one of the many on-line tutorials.[1] I will try to keep the code explanations as simple as possible, but it is beyond the scope of a few pages to describe *Verilog* in any detail. As with most programming languages, a few basic constructs go a long way. If you learn these few constructs, you can at least read and understand the code snippets. Also keep in mind that *Verilog* is used, among other things, to define the behavior of many different types of FPGAs or other hardware, write simulation code and design test benches. (A test bench is a virtual environment built to test the functionality of a piece of software or hardware.) Any *Verilog* skill that you pick up will be useful in understanding other FPGA programs that you encounter.

For hardware, you will need some kind of Altera FPGA development kit. To actually run the code that we are going to compile in this column, you will need a BeMicroSDK FPGA development kit and a UDPSDR-HF1 Receiver, (see Photo A) both available from Arrow Electronics.[2, 3] Even if you do not have the hardware, you can still follow along with the text and learn about FPGA coding for SDRs. Note that, while you will need some *Verilog* programming knowledge, advanced math skills and RF design experience are still absent. We are *analyzing* an existing SDR, not *designing* one from scratch.

For design software, we are in luck. Altera offers their *Quartus II* FPGA design software as a free download from the Internet for the FPGAs in their Cyclone® family of parts. Both *Linux* and *Windows* versions are available. The BeMicroSDK board contains an EP4CE22F17C7 part from the Altera Cyclone® IV E family of FPGAs, so we can use the free version of *Quartus II* design tools.

[1]Notes appear on page 18

Software Installation

After you have read up a bit on *Verilog* (or already grasp at least the basics), the next task is to download and install the *Quartus II* software. Go to the Altera web site (**www.altera.com**) under the **Design Tools & Services** tab and select **Design Software**.[4] Click on **Quartus II Web Edition Software** and then the **Download Web Edition Software – Free** button. Select release **14.0**, pick your operating system and download method (**Akamai DLM3 Download Manager** is faster, but is only available for *Windows* users) and make sure that you have selected the **Combined Files** tab. When you are sure that you have made your

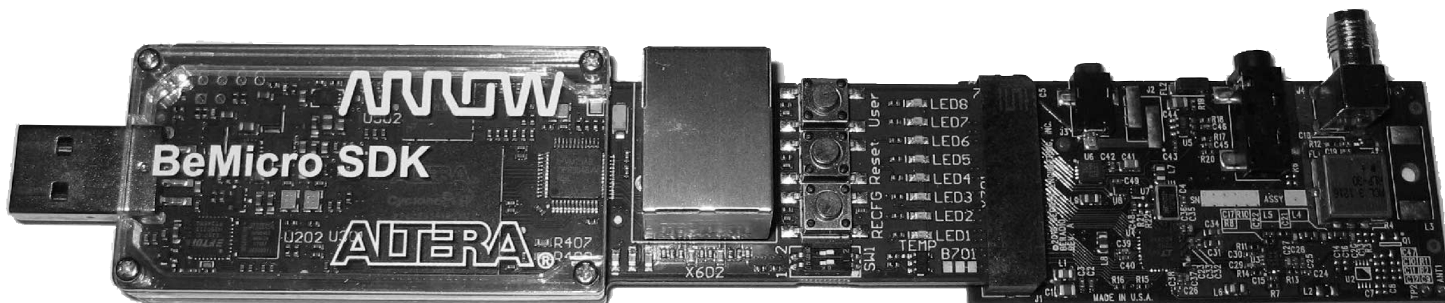

Photo A — The BeMicroSDK data engine together with the HF1 RF front-end board.

selections correctly, click on the blue down-arrow to begin the download. Be patient, the download is large at about 2.1 GB (*Windows*) or 4.0 GB (*Linux*), and may take some time if your connection is slow.

Note that the 14.0 (and newer) releases of *Quartus II* require a 64-bit operating system, *Windows 7* or later or *Red Hat Linux*. (I have successfully installed and run *Quartus II* 14.0 on 64-bit *Ubuntu Linux* 14.04LTS. If you are a *Linux* expert and are willing to read the Altera forums, you can likely make it work; it is beyond the scope of this column to help you with this!) If you do not have a 64-bit operating system, then you must download and install the previous version (13.1) of *Quartus II*. To run *Quartus II* version 13.1, you need *Windows XP* SP2 or later, 32- or 64-bit version. Simply select **13.1** in the **Select Release** dialog box before you begin your download. Don't worry if you need to run the older *Quartus II* version; the enhancements made to the newer 14.0 version do not really affect us when using the older Cyclone® IV parts.

A few notes on PC hardware are in order. Pretty much any PC that will run the SDR software (see the Sep/Oct 2014 *QEX* **Hands On SDR** column) will run the *Quartus II* software.[5] The most important hardware your *Quartus II* PC must have is memory. At least 2 GB is a minimum. Slower processors are okay (if you are willing to wait longer for compiles to finish), but stability of the software is not as good with less memory. Even simple FPGA compiles are significantly more compute intensive than compiling a "Hello World" program in *C*.

For help on getting *Quartus II* set up, and more hardware information on the BeMicroSDK, please take a look at the BeMicroSDK Embedded System Lab, modules 1, 2 and 3, available on line.[6] Don't worry that it was written for *Quartus II* version 12.1; with a few obvious adjustments, it is a good Lab to follow to gain more experience before we jump into our real SDR code.

To get a copy of the FPGA source code, download a copy of the *Quartus* archive from the SDRstick website.[7] The archive not only contains the source files (with a .v extension), but the pin assignment file (.qsf extension), timing constraints file (.sdc extension) and many other files needed to successfully compile the complete project. Once you have downloaded the archive file, start the *Quartus II* software and click on **<file><open project…>**. Navigate to the .qar file that you downloaded and click on it. From the dialog box that opens, select the destination folder (usually the default is fine) and click **OK**. *Quartus* will extract all of the files from the archive and set up the project, all ready to go.

Quartus II Quick Tips

While a *Quartus* tutorial is beyond the scope of this column, here are a few quick tips to get you started.[8] When you open *Quartus II*, you see a tool bar across the top of the window, and four "panes" within the window. See Figure 1. The upper left pane is the **Project Navigator** pane, below that is the **Tasks** pane, and across the bottom is the **Messages** pane. If you look closely, you will see these exact names in the title bar of each pane. The remaining upper right-hand pane is the **Workspace** area, where we will look at source code and report files, among other things.

At the bottom of the Project Navigator pane, there are several tabs: **Hierarchy**, **Files**, **Design Units** and so on. Click on the **Files** tab to see a list of all of the files in the project. You will see many *Verliog* source files (.v), a few ROM data files (.hex), a timing constraints file (.sdc) and a few others. Double-click on a Verilog source file in the **Project Navigator** pane and *Quartus* opens the file in the **Workspace** for you to view or edit. You can open as many files as you like; *Quartus* will make a tab in the **Workspace** for each file so you can switch quickly between them. Before we dig into specific sections of the code, let's take a look at the overall architecture of the FPGA firmware.

High-Level Overview

The FPGA RF processing is shown in Figure 2 and the audio processing is shown in Figure 3. The overall topology closely follows that of an analog direct conversion

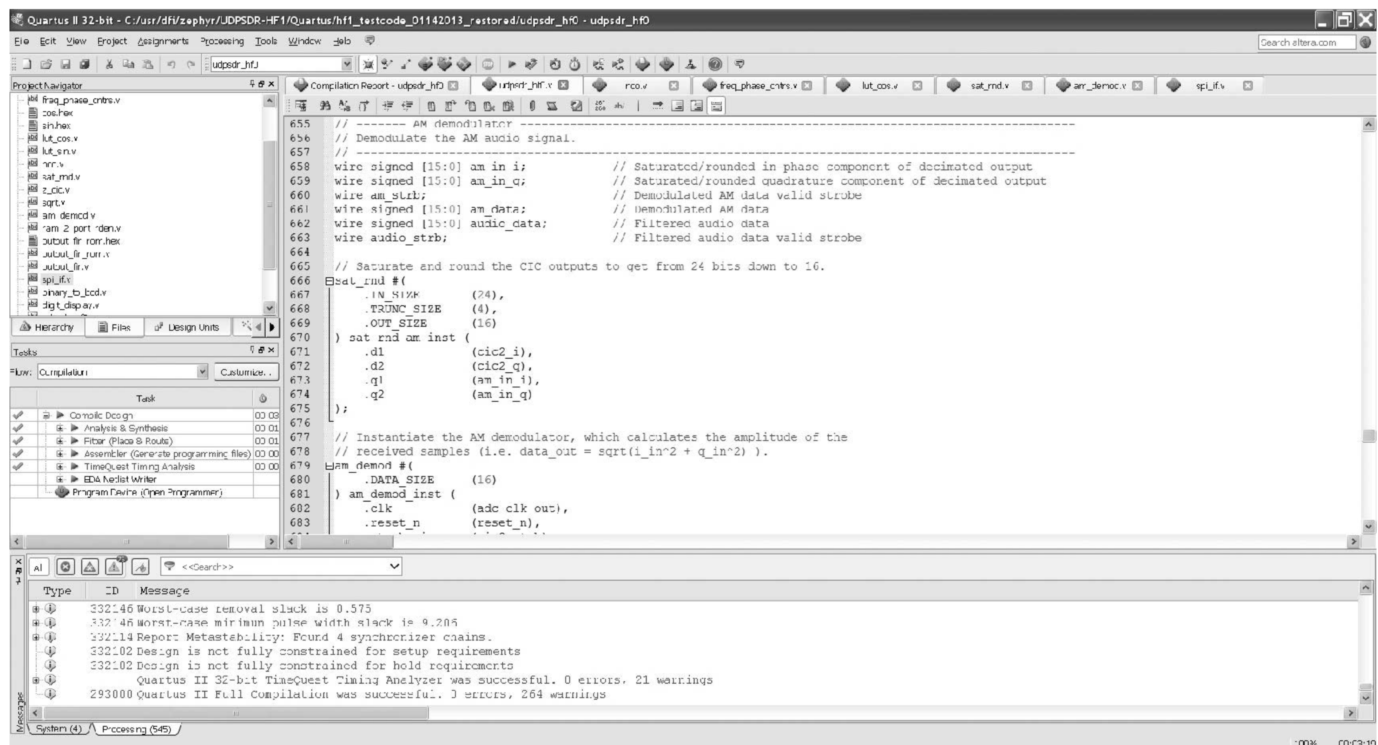

Figure 1 — This screen shot shows the *Quartus II* design software.

receiver. We start with two multipliers (quadrature mixer) fed by a Numerically Controlled Oscillator (NCO) acting as a digital local oscillator. The result is then filtered by two Cascaded Integrator-Comb (CIC) filters, demodulated by calculating the magnitude (square root of the sum of the squares), filtered again by a Finite Impulse Response (FIR) low-pass filter and scaled to provide a variable audio gain. The processed data is then clocked out to an audio DAC via a Serial Peripheral Interface (SPI) port. The SAT/RND blocks perform a saturation and rounding function to prevent overflow when we reduce the number of bits in the data path.

We will take an in-depth look at four of these blocks (multiplier, NCO, CIC and demodulator) and a cursory look at the rest of them. If you study the complete source code, you will discover that there are many more housekeeping and control functions that we do not cover. I have to leave something for you to figure out for yourself after you become a *Verilog* expert!

Even though I have reproduced small pieces of code here, you will find it helpful to open the "real thing" in the *Quartus* Workspace because the color formatting will make things easier to follow. To get started, open the *Verilog* source file **udpsdr_hf0.v** by double-clicking it. This file is the top-level file in the design. How do we know this? Take a quick look at the **Hierarchy** tab in the **Project Navigator**, and you will see it listed at the top level.

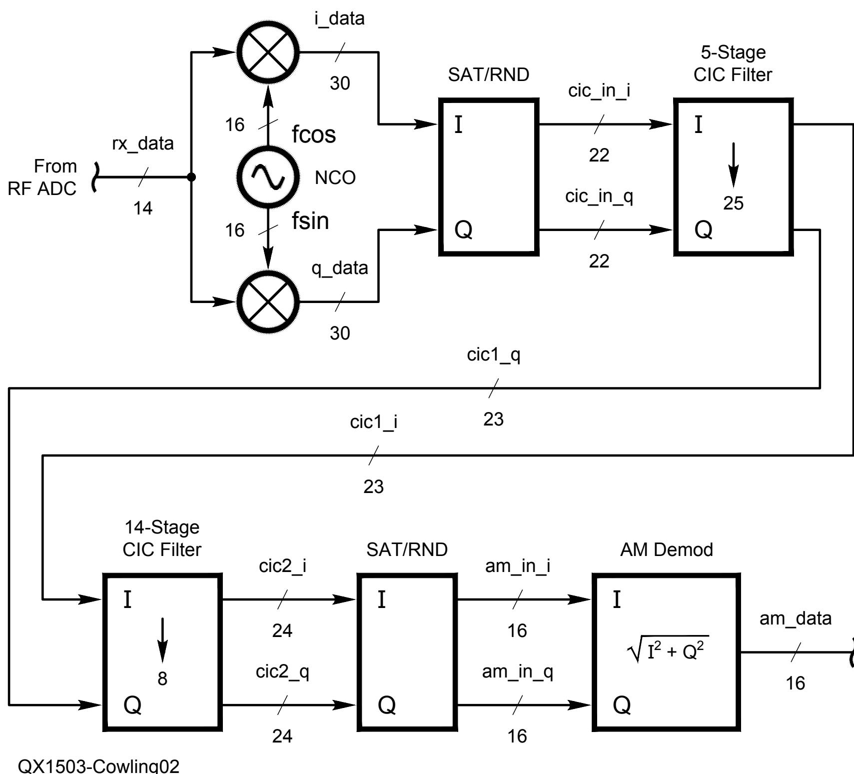

Figure 2 — HF1 FPGA test code RF section block diagram.

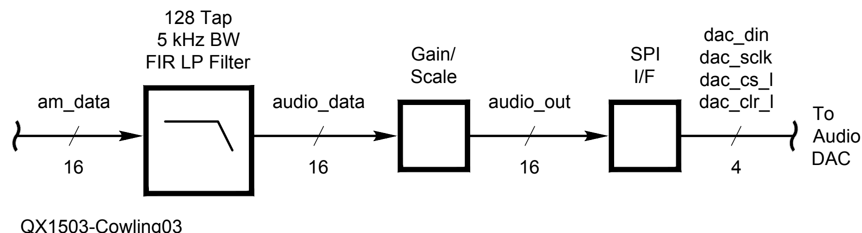

Figure 3 — HF1 FPGA test code AF section block diagram.

Quadrature Mixer

Scroll down to about line 570 in the **udpsdr_hf0.v** file that you just opened, or look at Figure 4 (line numbers in the file may be slightly different than in the figure). Anything after the "//" on a line is a comment, so lines 568 and 569 just document the function of the block. Lines 570 to 579 are called an "always block"; this block is a wrapper that contains some circuitry. In our case, this circuitry consists

Sidebar: HF0, HF1 and BeRadio

Throughout the *Verilog* code, you will see references to BeRadio, HF0 and HF1. While they seem to be interchangeable, they are not quite synonymous. Here is the 5-minute explanation of these terms. The SDRstick series of receivers originally consisted of three boards with three ascending performance levels: HF0 (12 bits at 10 Msps), HF1 (14 bits at 80 Msps) and HF2 (16 bits at 122.88 Msps). Since the HF0 was designed as a low-cost SDR demonstrator board to be used with BeMicroSDK, and other boards (such as the BeInMotion motor control board) already made use of the "Be" prefix, the name BeRadio was coined. BeRadio and HF0 are thus the exact same board.

BeRadio/HF0 was marketed by Arrow Electronics for only a limited time, and is no longer available. HF0 and HF1 are almost identical designs, however. They are so close, in fact, that they are built on the same base circuit board and differ only in the components that are soldered to the board. Such close hardware design kinship is the reason that HF0/BeRadio and HF1 can share large portions of their FPGA firmware. If you look closely, you can see that the *Quartus* **HF1_testcode** project was made from only slight modifications to the **HF0** project code.[15]

of two registers, one named **i_data** and the other named **q_data**. The width of these registers is defined in the code on lines 493 and 494 (not shown), but we can get a clue as to their width by looking at the widths defined in the initialization assignments on lines 572 and 573. The term **30'h0** means a value that is 30 bits wide, with a hexadecimal value of zero.

The list of edges and signals in parentheses immediately following the @ on line 570 is called the **sensitivity list**. In our case, whenever **adc_clk_out** rises from 0 to 1 or **reset_n** falls from 1 to 0, all of the statements within the always block are evaluated. The

```
568 // Shift received signal to zero by multiplying (mixing)
569 // with the local oscillator.
570 always @(posedge adc_clk_out or negedge reset_n) begin
571     if (!reset_n) begin
572         i_data <= 30'h0;
573         q_data <= 30'h0;
574     end
575     else begin
576         i_data <= fcos * rx_data;
577         q_data <= fsin * rx_data;
578     end
579 end
```

Figure 4 — This is a piece of the *Verilog* code for a quadrature mixer.

```
25 module z_nco (
26   input  wire clk,                        // System clock
27   input  wire reset_n,                    // System reset
28   input  wire [31:0] phase_inc,           // Phase increment
29   output wire [15:0] fcos,                // Cosine output
30   output wire [15:0] fsin                 // Sine output
31 );
32 // --------------------------------------------------------
33
34 // ---- Phase Accumulator ---------------------------------
35 reg [31:0] accum;                       // Phase accumulator
36
37 // Accumulate the current phase increment every clock cycle
38 always @(posedge clk or negedge reset_n) begin
39   if (!reset_n) begin
40      accum <= 32'h0;
41   end
42   else begin
43      accum <= accum + phase_inc;
44   end
45 end
46 // --------------------------------------------------------
47
48 // ---- Lookup Tables -------------------------------------
49 // Cosine lookup table.
50 lut_cos lut_cos_inst (
51   .address ( accum[31:20] ),
52   .clock ( clk ),
53   .q ( fcos[15:0] )
54 );
55
56 // Sine lookup table.
57 lut_sin lut_sin_inst (
58   .address ( accum[31:20] ),
59   .clock ( clk ),
60   .q ( fsin[15:0] )
61 );
62 // --------------------------------------------------------
63 endmodule
```

Figure 5 — This piece of *Verilog* code is for a numerically controlled oscillator (NCO).

begin keyword is used to define the boundary of the always block, and is matched with the **end** keyword on line 579. Every **begin** must have a matching **end**; they are paired just like parentheses. Note that the **end** statements are all indented to start in the same column as the beginning column of the line containing the matching **begin** keyword: line 574 aligns with line 571, line 578 aligns with line 575 and line 579 aligns with line 570. This is an example of good, easy to read *Verilog* coding style.

This always block defines the operation of the two 30-bit registers, **i_data** and **q_data**. The falling edge of **reset_n** (which is the assertion of reset, since **reset_n** is active low) causes both registers to be cleared (lines 571 to 574). Note that the **or** in the sensitivity list means that this happens irrespective of the **adc_clk_out** clock state, therefore making this reset an asynchronous one. The rising edge of **adc_clk_out** (as long as **reset_n** is de-asserted) will cause **i_data** to be updated with the value of the product of **fcos** and **rx_data**, while **q_data** is updated with the value of the product of **fsin** and **rx_data** (lines 575 to 578). This is our quadrature "mixer": two numerical multipliers.

Note the widths of the inputs to each multiplier. (You can look in the code on lines 444, 488 and 489 or look on the block diagram in Figure 2.) When we multiply, the bit width of the output is the sum of the widths of the inputs, so we must make sure that the variable that is assigned the product is defined to be wide enough. This is a signed multiply because both inputs and outputs are declared as signed numbers. If you forget to declare these as signed numbers, the *Verilog* compiler will implement an unsigned multiplier, which is a common *Verilog* coding error to be avoided. One other thing to note is the order of the assignments in the always block. The reset code is implemented as the first part of an **if…else** construct to ensure that the asynchronous reset takes precedence over the synchronous multiply operation. However, all variable values within the block are updated at the same time, regardless of the order of the assignment statements. All 60 bits of **i_data** and **q_data** are updated simultaneously in parallel (whether set to zero by **reset_n** or to the products of other variables by **adc_clock_out**). I have reviewed this simple block of code in detail because it is the first one. We will move a bit faster on the next blocks, focusing more on what the block does rather than how *Verilog* works.

Numerically Controlled Oscillator (NCO)

The next block that we will analyze is the numerically controlled oscillator, or NCO. The NCO is a bit different than a simple oscillator in that it produces two sine wave outputs that are 90° out of phase. The first output is named **fcos** and will be used to calculate **i_data** values. The other output is named **fsin** and will be used to calculate **q_data** values. These variable names should already be familiar, as they are used by the quadrature mixer discussed earlier.

The behavior of the **z_nco** module is defined in the **nco.v** file, the majority of which is reproduced in Figure 5. The module is *instantiated* in the main file (see lines 560 to 566 of **udpsdr_hf0.v**) in much the same way a component is placed on a schematic: call out the module name, give it a unique instance name (like a reference designator on a schematic, for example R22 or U3) and connect up inputs and outputs to the module. The direction, type, width and name of module I/O signals are defined on lines 26 to 30.

The NCO outputs are derived from two lookup tables, **lut_cos** and **lut_sin**. A phase accumulator named **accum** is used as an address into the lookup tables. The accumulator is incremented on every clock cycle by a number of counts determined by the value of the module input, **phase_inc**. The larger the value of **phase_inc**, the faster the lookup tables are "scanned," and therefore the higher the NCO output frequency. You can take a look at the 16-bit values in the lookup tables by opening the **cos.hex** and **sin.hex** files in the **files** tab of the *Quartus* Project Navigator pane (make sure you specify 16 bits as the width when asked). You might ask, "How did you create the two look up table files?" Well, that is a very good question. We actually wrote a small program in *Python* to calculate the values in the two hex files. We then used a *Quartus* memory generator wizard to use the hex files to initialize two 4096 × 16 SRAM blocks as read-only memories (ROMs). The two ROMS become our look-up tables. Take a look at the wizard-generated **lut_cos.v** and **lut_sin.v** files and you can probably figure out how we did it.

The astute reader is probably wondering why we used two ROMs instead of just offsetting the address into one ROM to achieve the desired 90° phase shift. (You sure are full of good questions today!) The answer is that the logic is simpler and we are lazy. Remember that both lookup tables are accessed every clock cycle. If you want two 16-bit numbers (one for sine and one for cosine) every clock cycle from the same ROM, then you have to read it twice as fast as the clock. While this is possible, it is not as simple as just using more memory. After all, that 16 K bytes of memory is not being used for anything else… (Now I am talking like a software guy.)

Saturate and Round Module

If you would like to explore this function in detail, open the **sat_rnd.v** file by double-clicking it in the *Quartus* Project Navigator **files** tab. This is a *parameterizable* module,

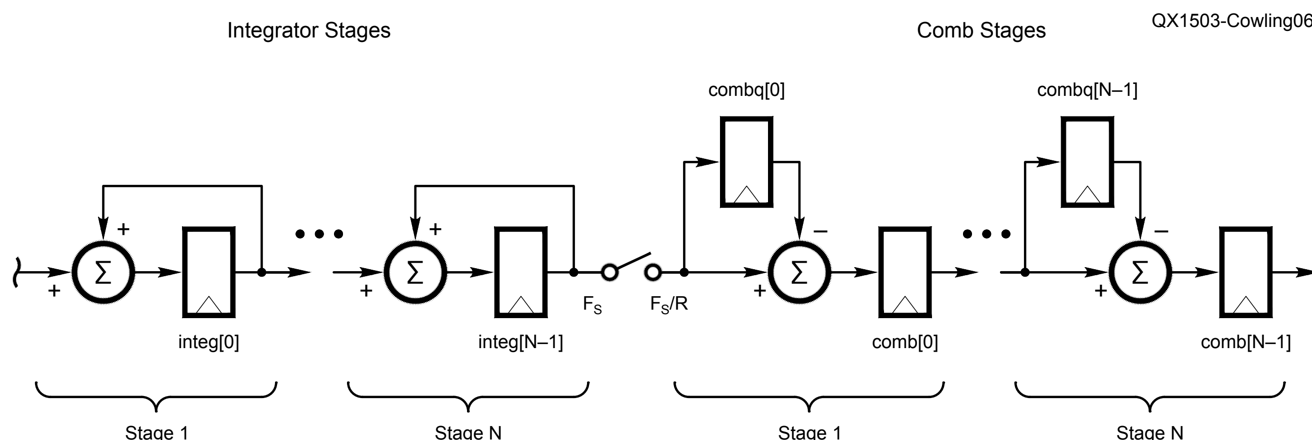

Figure 6 — Here is a pipelined CIC decimating filter block diagram.

```
17  module z_cic
18  #(
19      parameter IN_SIZE  = 16,                                   // Input data width
20      parameter OUT_SIZE = 16,                                   // Output data width
21      parameter N_STAGES = 5,                                    // Number of stages
22      parameter DEC_RATE = 10                                    // Decimation rate
23  ) (
24      input    wire  clk,                                        // System clock
25      input    wire  reset_n,                                    // System reset
26      input    wire  instrobe,                                   // Input sample valid strobe
27      input    wire  signed [IN_SIZE-1:0] in1_data,              // Channel 1 input sample
28      input    wire  signed [IN_SIZE-1:0] in2_data,              // Channel 2 input sample
29      output   wire  outstrobe,                                  // Output sample valid strobe
30      output   reg   signed [OUT_SIZE-1:0] out1_data,    // Channel 1 output sample
31      output   reg   signed [OUT_SIZE-1:0] out2_data     // Channel 2 output sample
32  );
33  // -----------------------------------------------------------------------------------------------
34
35  // ---- Function Definitions --------------------------------------------------------------------
36  // Function to calculate ceiling of Log base 2 of a value.
37  function integer clog_b2;
38      input [31:0] value;
39      integer tmp;
40      begin
41          tmp = value - 1;
42          for (clog_b2 = 0; tmp > 0; clog_b2 = clog_b2 + 1) tmp = tmp >> 1;
43      end
44  endfunction
45  // -----------------------------------------------------------------------------------------------
46
47  // ---- User Parameters -------------------------------------------------------------------------
48  // Derive internal parameters from input parameters using the Log2 function.
49  // -----------------------------------------------------------------------------------------------
50 localparam CNTR_SIZE = clog_b2(DEC_RATE);              //Size of sample decimation counter
51 localparam ACC_SIZE  = IN_SIZE + (N_STAGES*CNTR_SIZE);  //Width of integration accumulators
52// -------------------------------------------------------------------------------------------------
53
54  // ---- Module Control --------------------------------------------------------------------------
55  reg [CNTR_SIZE-1:0] sample_count;                              // Sample decimation counter
56  reg combstrobe;                                                // Strobe for activating comb stages
57  reg [1:0] del_strobe;                                          // Pipelined comb strobe to match
latency
58
59  // Generate internal strobe for every DEC_RATE input strobes.
60  always @(posedge clk or negedge reset_n) begin
61      if (!reset_n) begin
62          sample_count <= {(CNTR_SIZE){1'b0}};
63      end
64      else begin
```

Figure 7 — Part 1 of the *Verilog* code for a cascaded integrator-comb filter.

```
65          del_strobe <= {del_strobe[0] , combstrobe};
66          if (instrobe) begin
67              if (sample_count == DEC_RATE - 1) begin
68                  sample_count <= {(CNTR_SIZE){1'b0}};
69                  combstrobe <= 1'b1;
70              end
71              else begin
72                  sample_count <= sample_count + 1'b1;
73                  combstrobe <= 1'b0;
74              end
75          end
76          else begin
77              combstrobe <= 1'b0;
78          end
79      end
80 end
81  // ------------------------------------------------------------------------------------------
82
83  // ---- Integrator Stages -------------------------------------------------------------------
84  reg signed [ACC_SIZE-1:0] integ1 [N_STAGES-1:0];             // Array of integrators for channel 1
85  reg signed [ACC_SIZE-1:0] integ2 [N_STAGES-1:0];             // Array of integrators for channel 2
86  integer i;                                                   // FOR loop variable
87
88  // For each integration stage, integrate the value of the previous stage. The
89  // first stage integrates the input data.
90  always @(posedge clk or negedge reset_n) begin
91      if (!reset_n) begin
92          for (i = 0; i < N_STAGES; i = i + 1) begin
93             integ1[i] <= 0;
94             integ2[i] <= 0;
95          end
96      end
97      else begin
98          if (instrobe) begin
99              integ1[0]      <= integ1[0] + {{(ACC_SIZE-IN_SIZE){in1_data[IN_SIZE-1]}},in1_data};
100             integ2[0]      <= integ2[0] + {{(ACC_SIZE-IN_SIZE){in2_data[IN_SIZE-1]}},in2_data};
101             for (i = 1; i < N_STAGES; i = i + 1) begin
102                 integ1[i]   <= integ1[i] + integ1[i-1];
103                 integ2[i]   <= integ2[i] + integ2[i-1];
104             end
105         end
106     end
107 end
```

meaning that certain characteristics of the module can be defined when it is instantiated by setting values of pre-defined parameters. These parameter values are used by the code to modify the way the module behaves. Take a look at the block diagram in Figure 2. Notice that the SAT/RND module is used twice, once before the first CIC filter and once after the second CIC filter. In the former case, it reduces the data width from 30 to 24 bits and in the latter case, from 24 to 16 bits. (See **udpsdr_hf0.v** lines 605 to 617 and 666 to 675 for the two instantiations.) I will explain how it does this after I describe the function of this module.

The SAT/RND module performs three operations on its input data. First it truncates the data to a specific bit width by removing a number of least-significant bits from the input data. Next it rounds the result to the nearest value based on the discarded bits. Finally, it checks to make sure that the rounded value can be represented properly in the output bit width (in other words, there is no overflow or underflow). Note that the input and output data are signed numbers. If overflow is detected, the output is set to the maximum positive value (sign bit is 0, all other bits are 1). If underflow is detected, the maximum negative value is used instead (sign bit is 1, all other bits are 0). These numbers are called saturation values.

When the module is instantiated, parameters IN_SIZE, OUT_SIZE and TRUNC_SIZE are specified corresponding to the input bit width, output bit width and the number of bits to remove, respectively. The same module is used in both places in our block diagram, but each is instantiated with different values for the three parameters. This is useful, since we save ourselves the work of writing two different modules. Of course, TANSTAAFL (yes, it is more than three letters, and yes, I am going to make you turn to the end of the column to look it up),[9] so we end up with a slightly more complex module as a result. This sure seems like a lot of trouble just to reduce the number of bits in the data stream, but it is essential to minimize overflow and underflow discontinuities.

Cascaded Integrator-Comb (CIC) Filters

I have deliberately given you less and less assistance in analyzing the *Verilog* code in the last three sections. The goal is for you to eventually be able to read and digest new sequences of code on your own. This next section will test your skill (and perhaps your patience, too) with yet more complexity. Take a look at Figure 6, the block diagram of the CIC filter. Each circle containing a Σ represents an adder and each block is a register. An adder-register pair is called an *accumulator*. I have shown only two stages of the filter (first and last) for brevity, but the ellipsis shows where additional stages are inserted. The switch symbol shown between the integrator stages and the comb stages represents a reduction in the clock rate by a factor equal to the decimation rate, R. Thus, the left-most CIC filter shown in Figure 2 (5-stages) consists of ten accumulators and five registers for each data path (i and q), along with the register clocking circuitry, which I will explain shortly.

This is a parameterized design, with input and output bit widths, number of stages and the decimation rate set at instantiation time. The numbers given on lines 19 to 22 of Figure 7 are the defaults that are used if one or more parameters are not set at instantiation. You can look in **udpsdr_hf0.v** lines 621 to 624 and 638 to 641 for the instantiated values for the 5-stage and 14-stage CIC filters, respectively. The 5-stage filter uses an **IN_SIZE** of 22, **OUT_SIZE** of 23, **N_STAGES** of 5 and **DEC_RATE** of 25.

Let's look at the register and accumulator structure in the code before we see how the registers are clocked. The integration stage accumulator register arrays **integ1** and **integ2** are defined on lines 84 and 85 of Figure 7. The bit range (register width) is defined in the left-hand set of square brackets. The array index range is defined in the right-hand set of brackets. Notice that the number of bits in these accumulator registers is wider than the input bit width since we need to hold the sum of many input samples. (See line 51 for the definition of **ACC_SIZE**.) The number of registers in each array is equal to the number of stages in the filter. The behavior of the accumulators is defined within the always block on lines 90 to 107 of Figure 7. The first accumulator adds the sign-extended input data to its current value on each clock. The remaining accumulators each sum the output from the previous accumulator and their own current value on each clock using the **for** loop on lines 101 to 104.

The **comb** and **combq** registers are defined on lines 111 to 114 of Figure 8. The comb registers are used as registers in the accumulators (they each directly follow an adder), while the combq registers store the value of each accumulator's (+) input, to be used one clock cycle later at the accumulator's (–) input. The always block on lines 151 to 162 performs one last function that is not shown in the block diagram of Figure 6: it truncates the output to the number of bits specified by the **OUT_SIZE** parameter and rounds to the nearest bit.

Notice that every one of the four always blocks in Figures 7 and 8 are clocked by the **clk** input clock. It is much easier to analyze the register timing of this synchronous design than it would be if the output registers were clocked by a different clock signal. So where does the decimation occur, then? Remember that the output data rate is equal to the input rate divided by the decimation rate. If you look at Figure 8, line 130 you will see that while the always block is evaluated on every **clk** edge, the **if** condition will be true only when **combstrobe** is true. Take a look at the always block in Figure 7, lines 60 to 80; this is where **combstrobe** is generated. With a bit of study, you can see that one **combstrobe** is generated for every **DEC_RATE instrobe** assertions. The output rate will therefore be slower than the input rate by a factor equal to the decimation rate.

Altera's AN455 application note is an excellent place to start for more information on CIC filters in FPGAs.[10]

AM Demodulator

The AM demodulator code is in the **am_demod.v** file, most of which is reproduced in Figure 9. The magnitude of the AM demodulator output is equal to the square root of the sum of the squares of the i and q input components. The always block on lines 40 to 51 squares the incoming i and q values. The assign statement on line 55 then adds them together, while the always block on lines 76 to 83 takes the square root of the sum. Note that we delay the **strobe_in** signal to account for the number of clock cycles that are required to calculate the magnitude, and then assign the delayed signal to the **strobe_out** of the module on line 86. This is called *pipelining* the signal.

Finite Impulse Response (FIR) Low-pass Filter

While explaining the coding of an FIR filter is beyond the scope of this column, there are good references on the Internet.[11, 12] Basically, the FIR filter is just a digital low-pass filter consisting of a series of multipliers and registers feeding a large adder tree. The FIR filter requires a table of coefficients that are typically supplied by a ROM, in much the same way that the numerically controlled oscillator stores its lookup tables. Altera provides a software wizard to assist you in calculating the coefficients. The number of multipliers, called taps, plus the values of the coefficients, determines the LPF cutoff frequency and its slope. The FIR filter code is in the file **output_fir.v**, which also uses the three files **output_fir_rom.v**, **output_fir_rom.hex** and **ram_2_port_rden.v**.

Audio Gain, Scaling and Serial Peripheral Interface (SPI)

The filtered audio is multiplied by an audio gain coefficient to set the desired

```
110 // ---- Comb Stages ------------------------------------------------------------------------------
111 reg signed [ACC_SIZE-1:0] comb1 [N_STAGES-1:0];    // Array of comb stages for channel 1
112 reg signed [ACC_SIZE-1:0] comb1q [N_STAGES-1:0];   // Array of delayed comb values for channel 1
113 reg signed [ACC_SIZE-1:0] comb2 [N_STAGES-1:0];    // Array of comb stages for channel 2
114 reg signed [ACC_SIZE-1:0] comb2q [N_STAGES-1:0];   // Array of delayed comb values for channel 2
115 integer j;                                         // FOR loop variable
116
117 // For each comb stage, subtract the previous value of the previous stage from
118 // the current value of the previous stage. The first stage subtracts from the
119 // value of the final integration stage.
120 always @(posedge clk or negedge reset_n) begin
121     if (!reset_n) begin
122         for (j = 0; j < N_STAGES; j = j + 1) begin
123             comb1[j]  <= 0;
124             comb1q[j] <= 0;
125             comb2[j]  <= 0;
126             comb2q[j] <= 0;
127         end
128     end
129     else begin
130         if (combstrobe) begin
131             comb1[0]       <= integ1[N_STAGES-1] - comb1q[0];
132             comb1q[0]      <= integ1[N_STAGES-1];
133             comb2[0]       <= integ2[N_STAGES-1] - comb2q[0];
134             comb2q[0]      <= integ2[N_STAGES-1];
135            for (j = 1; j < N_STAGES; j = j + 1) begin
136                 comb1[j]   <= comb1[j-1] - comb1q[j];
137                 comb1q[j]  <= comb1[j-1];
138                 comb2[j]   <= comb2[j-1] - comb2q[j];
139                 comb2q[j]  <= comb2[j-1];
140             end
141         end
142     end
143 end
144 // ---------------------------------------------------------------------------------------------
145
146 // ---- Output ---------------------------------------------------------------------------------
147 // Assign final element of delayed comb strobe as the output strobe.
148 assign outstrobe = del_strobe[1];
149
150 // Round off LSBs of final comb output to get filter output.
151 always @(posedge clk or negedge reset_n) begin
152     if (!reset_n) begin
153         out1_data <= 0;
154         out2_data <= 0;
155     end
156     else begin
157         out1_data <= comb1[N_STAGES-1][ACC_SIZE-1:ACC_SIZE-OUT_SIZE] +
158                      comb1[N_STAGES-1][ACC_SIZE-OUT_SIZE-1];
159         out2_data <= comb2[N_STAGES-1][ACC_SIZE-1:ACC_SIZE-OUT_SIZE] +
160                      comb2[N_STAGES-1][ACC_SIZE-OUT_SIZE-1];
161     end
162 end
163 // ---------------------------------------------------------------------------------------------
164 endmodule
```

Figure 8 — Part 2 of the *Verilog* code for a cascaded integrator-comb filter.

volume. Selected bits of the result are formatted and serially shifted out to match the serial peripheral interface (SPI) of the audio DAC on the HF1 board. The bits are selected to provide the loudest volume while still preventing DAC overload.

Perceptive readers will note that this code actually contains a NIOS II soft-core CPU to control some functions. We have deliberately avoided adding the complexity of an embedded CPU to our discussion, leaving that topic instead for another day. If you are ambitious, the entire source for the NIOS II CPU is included in the ***Quartus*** Archive (.qar file) for your amusement.

What's Next?

Now that you have a working knowledge of FPGA techniques for SDR, what can you do next? The openHPSDR project is open source, so why not take a look at the FPGA code for the Mercury receiver, Pennylane transmitter, Metis Ethernet interface or even the Hermes transceiver? Each one of these boards has an on-board FPGA and *Verilog* code to match. It is all available from the openHPSDR repository, and you are now qualified to download it, read it, understand it and even modify, compile and run it on your own HPSDR hardware if you like.[13, 14] The tools that you have used today are the very same tools that the developers use when they write or update the code.

Next time we can cover how to compile, download and execute code in the BeMicroSDK FPGA, or we can go off in another direction, such as GNU Radio. Please drop me an e-mail if you have any suggestions for topics you would like to see covered in future Hands-On-SDR columns or even just to let me know whether or not you found this discussion useful.

Notes

[1]Many *Verilog* tutorials and references are available by searching "Verilog tutorial" with your favorite search engine. Here are a few links:
Tutorial: **doulos.com/knowhow/verilog_designers_guide/**.
Tutorial: **vol.verilog.com/VOL/main.htm**.
Reference: **sutherland-hdl.com/online_verilog_ref_guide/vlog_ref_top.html**.
Reference: **see.ed.ac.uk/~gerard/Teach/Verilog/manual/**.

[2]The BeMicroSDK development kit circuit board is available from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/bemicrosdk**.

[3]The UDPSDR-HF1 development kit circuit board is available from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/udpsdr-hf1**.

[4]You can download the free Altera Web Edition software from the Altera website: **altera.com/products/software/quartus-ii/web-edition/qts-we-index.html**.

[5]Scotty Cowling, WA2DFI, "Hands On SDR," *QEX*, Sep/Oct 2014, pp 31 – 38.

[6]Download the Altera BeMicroSDK embedded system lab document: **download.siliconexpert.com/pdfs/2013/5/20/12/17/55/611/arrowd_/manual/bemicro_sdk_embedded_system_hw_lab_qsys_v12_1.pdf**.

[7]The source code is available from the SDRstick SVN at **svn.sdrstick.com** under the <**sdrstick-release/BeMicroSDK/udpsdr-hf1/firmware/source**> directory. The file name is <**hf1_testcode_11182014.qar**>

[8]Introduction to *Quartus II* Software: <**altera.com/literature/manual/quartus2_introduction.pdf**>

[9]TANSTAAFL, or "There ain't no such thing as a free lunch" has several popular usages, including in science fiction and economics. See <**en.wikipedia.org/wiki/There_ain%27t_no_such_thing_as_a_free_lunch**>

[10]Altera AN455, "Understanding CIC Compensation Filters," <**altera.com/literature/an/an455.pdf**>

[11]FIR Filter Design from Altera Wiki: <**alterawiki.com/wiki/FIR_Filter_Design_in_Arria_V/Cyclone_V_DSP_Block_Using_VHDL_Inferring**>

[12]Implementing FIR Filters and FFTs, Altera white paper: <**altera.com/literature/wp/wp-01140-fir-fft-dsp.pdf**>

[13]Look in the TAPR repository **svn.tapr.org** in <main/trunk> under the board name

[14]The openHPSDR hardware is available from TAPR at **tapr.org/hpsdr_index.html**

[15]The HF0 source code can be found at **svn.sdrstick.com** in the <**sdrstick-release/beradio/beradio-firmware/source**> directory. The file name is <**BeRadio_lab_01232013.qar**>

```
15 module am_demod
16 #(
17     parameter DATA_SIZE = 16                     // Bits in data path
18 ) (
19     input   wire  clk,                               // System clock
20     input   wire  reset_n,                           // Asynchronous system reset
21     input   wire  strobe_in,                         // Input data valid strobe
22     input   wire  signed  [DATA_SIZE-1:0] i_in,      // In-phase input data
23     input   wire  signed  [DATA_SIZE-1:0] q_in,      // Quadrature input data
24     output  wire  strobe_out,                        // Output data valid strobe
25     output  wire  signed  [DATA_SIZE-1:0] data_out   // Output data
26 );
27 // ------------------------------------------------------------------------------------
28
29 // ---- AM Demodulator ----------------------------------------------------------------
30 reg  [2:0] strb_sr;                      // Shift register to pipeline input strobe
31 reg  signed [2*DATA_SIZE-1:0] i_sqrq;   // Squared in-phase data
32 reg  signed [2*DATA_SIZE-1:0] q_sqrq;   // Squared quadrature data
33 wire [2*DATA_SIZE:0] sqrsum;            // Sum of squares of in-phase and quadrature data
34 reg  [2*DATA_SIZE-1:0] sqrsumq;         // Registered sum of squares
35 wire [DATA_SIZE-1:0] sqrt_data;         // Amplitude of received data
36 reg  signed [DATA_SIZE-1:0] sqrtq;      // Registered amplitude of received data
```

Figure 9 — Here is a sample piece of *Verilog* code for an AM demodulator.

```
37
38 // Square the in-phase and quadrature components of the input.
39 // Pipeline the input strobe to match latency.
40 always @(posedge clk or negedge reset_n) begin
41     if (!reset_n) begin
42         strb_sr    <= 3'h0;
43         i_sqrq     <= {(2*DATA_SIZE){1'b0}};
44         q_sqrq     <= {(2*DATA_SIZE){1'b0}};
45     end
46     else begin
47         strb_sr    <= {strb_sr[1:0], strobe_in};
48         i_sqrq     <= i_in * i_in;
49         q_sqrq     <= q_in * q_in;
50     end
51 end
52
53 // Sum of the squares plus one to implement rounding. If bit 0 is set, this
54 // rounds up (more positive).
55 assign sqrsum = i_sqrq + q_sqrq + {{(2*DATA_SIZE-1){1'b0}},1'b1};
56
57 // Register the sum of the squares after rounding off the LSB.
58 always @(posedge clk or negedge reset_n) begin
59     if (!reset_n) begin
60         sqrsumq    <= {(2*DATA_SIZE){1'b0}};
61     end
62     else begin
63         sqrsumq    <= sqrsum[2*DATA_SIZE:1];
64     end
65 end
66
67 // Calculate the amplitude of the received signal as the square root of the
68 // sum of the squares.
69 sqrt sqrt_inst (
70     .radical (sqrsumq),
71     .q (sqrt_data),
72     .remainder ()
73 );
74
75 // Register the square root output.
76 always @(posedge clk or negedge reset_n) begin
77     if (!reset_n) begin
78         sqrtq <= {DATA_SIZE{1'b0}};
79     end
80     else if (strb_sr[1]) begin
81         sqrtq <= sqrt_data;
82     end
83 end
84
85 // Assign output strobe and data.
86 assign strobe_out = strb_sr[2];
87 assign data_out   = sqrtq;
88 // ------------------------------------------------------------------------------------------
89 endmodule
```